Laboratory Report 1

Genetic and Evolutionary Algorithms

Submitted by:
Wiktor Koziak
w.koziak@aol.com
279734

March 20, 2017

1. Introduction

The main goal of the laboratory was to create a basic genetic algorithm. According to the instructions the algorithm would consist of the following steps:

I. Population initialization

The population is initialized as an array of chromosomes (later referred to as individuals). Each chromosome consists of *n* parameters (coded as binary values: 1 or 0). The values of parameters are generated randomly.

II. Evaluation of the population

The individuals are evaluated by the fitness function. In this example a very basic method was used: for each individual the sum of parameters is returned by the fitness function. The array consisting of fitness values is then sorted, having indices of individuals retained.

III. Selection for mating

From the whole set of individuals parents are selected for mating. The selection is a fitness proportionate selection, which favours the individuals of greatest fitness. From selected individuals a parent array is formed.

IV. Cross-over

The selected parents are then crossed-over with a simple swap function, which replaces part of "genetic code" of a parent with corresponding part of the code of the second parent.

V. Evaluation of the population

The population is evaluated again the same way it was in step II.

VI. Repetition of steps III – V until stopping condition is satisfied

Stopping condition in this example is reaching a certain value of iterations.

The algorithm will be presented first as code snippets covered in detail in separate points. The whole code can be found in Appendix 1, 2 and 3.

2. Parameters of algorithm

For this particular case the following parameters were used in the algorithm:

Number of values each bit of a chromosome can have:	`n_param = 1`
Number of bits in a chromosome:	`n_elem  = 10`
Population size:	`n_param = 20`

3. Population initialization

The population of given size is initialized with the function `randi()`, which generates random integers from given range in a matrix of given size:

```
n_param = 1;       % number of values per bit
n_elem = 10;       % number of bits in chromosome
population = 20;   % size of population

if rem(population, 2) ~= 0
    warning('Population size should be even');
end

x = randi([0 1], [population n_elem]);    % population, values 0 or 1
```

Fig. 1 Population initialization

4. Evaluation of the population

As mentioned before the individuals are evaluated by a fitness function which simply calculates the sum of values in each chromosome.

```
ff = sum(x, 2);                % cost of population
ff_total = sum(ff);            % sum of fitness
average_ff = mean(ff)          % average fitness
```

Fig. 2 Population evaluation

5. Selection for mating

The parents for mating are selected with proportionate selection algorithm, which tends to select individuals with greater fitness rather than those with lower fitness. `matingVector` has the size of the population matrix, thus selects the same amount of individuals for mating (though it doesn't select every individual – individuals may repeat within the parent matrix).

The selection algorithm is written within a separate function file (`select_individuals.m`).

```
for j = 1:length(probability)
        matingVector(j,1) = sum(probability(1:j));
    end

    for j = 1:length(probability)
        matingSelector = rand;

        for i = 1:length(probability)
            if matingSelector < matingVector(i,1)
                selectedIndividuals(j) = i;
                break
            end
        end
    end
```

Fig. 3 Selection algorithm

6. Cross-over

For each 2 parents the procedure described in introduction is performed. The code in Fig. 4 repeats until all individuals from parents' matrix have mated.

The cross-over algorithm is written within a separate function file (`mate.m`).

```
crossPoint = round(rand*nelem);

        Indi1 = [x(I(selectedIndividuals(i)), 1:crossPoint)
x(I(selectedIndividuals(i-1)), crossPoint+1:nelem)];
        Indi2 = [x(I(selectedIndividuals(i-1)), 1:crossPoint)
x(I(selectedIndividuals(i)), crossPoint+1:nelem)];

        new_population(population + 1 - i, :) = Indi1;
        new_population(population + 2 - i, :) = Indi2;
```

Fig. 4 Cross-over algorithm

7. Re-evaluation of population and program loop

The population is re-evaluated after each cross-over procedure to give information about changes that occurred. Selection and cross-over steps are repeated until the stopping condition is satisfied, which in this case is number of iterations, but can easily be changed for e.g. maximum fitness or stagnation point achieved.

```
while iterations < 70

    [B,I] = sort(ff); % B-fitness sorted, I-corresponding indices

    selectedIndividuals = select_individuals(B);

    new_x = mate(selectedIndividuals, x, I, population);
    x = new_x;

    ff = sum(x, 2);           % re-evaluation
    ff_total = sum(ff);

    iterations = iterations + 1;

end
```

Fig. 5 Main program loop

8. Results

Below are presented several figures with the plots showing change of average fitness in population over iterations (top) and maximum fitness over iterations (bottom). Each figure comes from separate program run. The results are discussed in point 9.

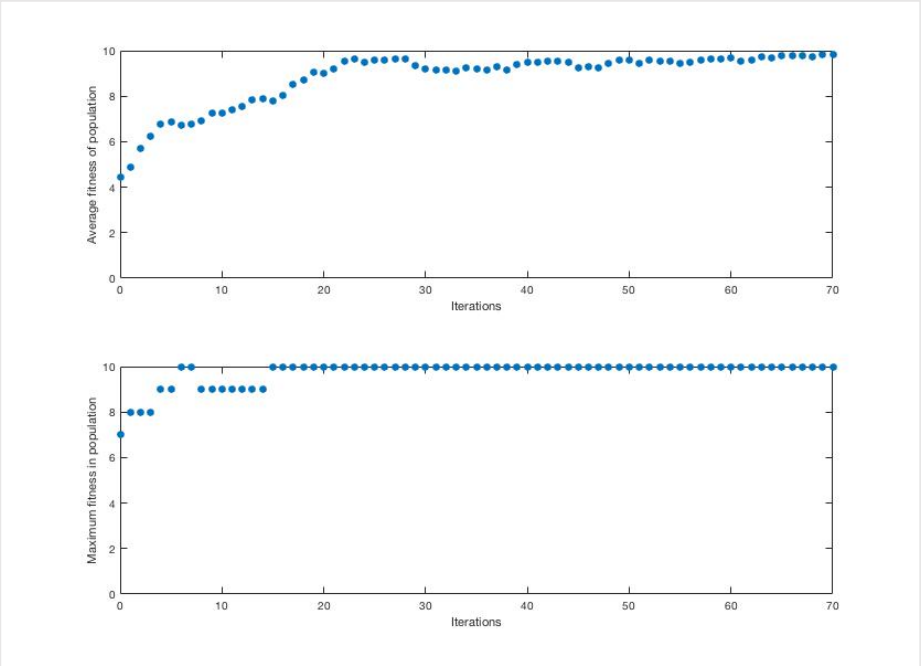

Fig. 6 Result 1

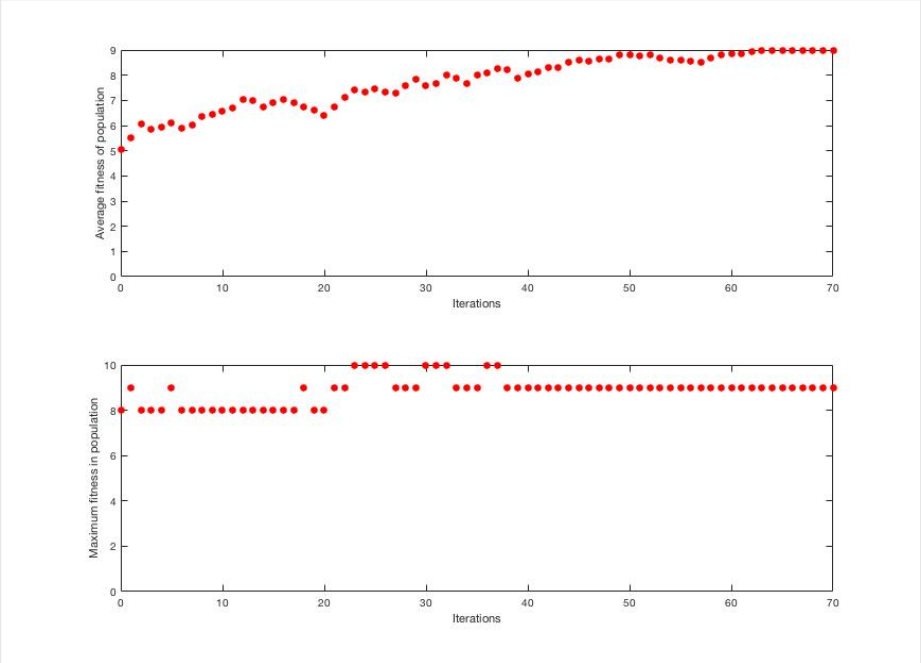

Fig. 7 Result 2 (failed)

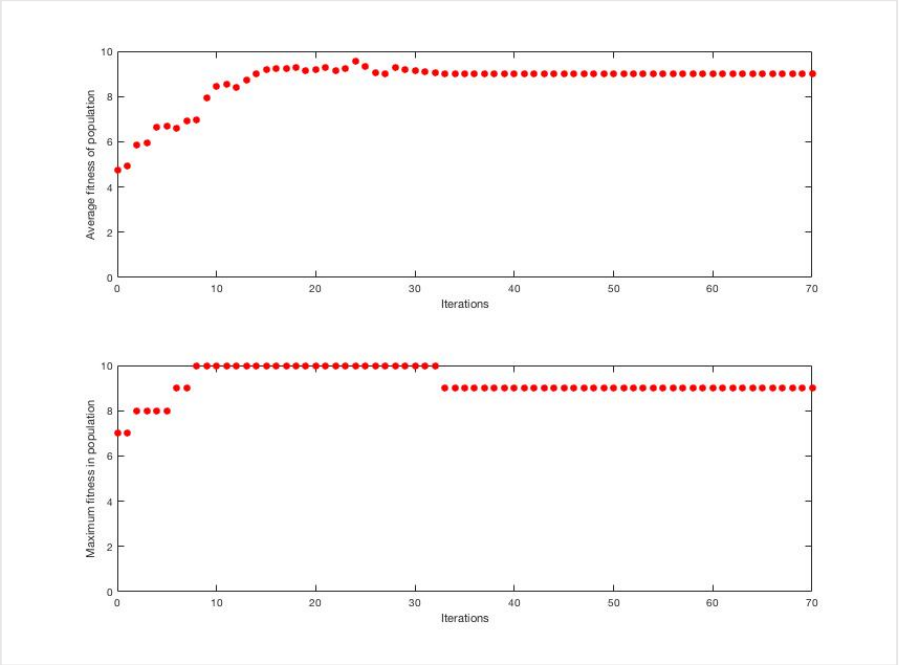

Fig. 8 Result 3 (failed)

9. Discussion of the results

As shown in the figures 6-8 the algorithm does not always succeed to achieve maximum average fitness (which means every individual has the same, maximum fitness). In some cases, even over 22,000 iterations were not enough to achieve the goal, therefore one can safely assume some parts of the code need improvement, most probably the selection or mating algorithm.

10. Future improvements

In the next versions of the algorithm changes in how individuals are selected will be made. A cross-over probability will be implemented, as well as possibility of mutation of individual genes.

Appendix 1 (genetic-algorithm-1.m)

```
clear all;
clc;

n_param = 1;       % number of parameters in chromosome
n_elem = 10;       % number of bits in each parameter
population = 20;   % size of population

if rem(population, 2) ~= 0
    warning('Population size should be even');
end

x = randi([0 1], [population n_elem]);    % population, values 0 or 1

ff_init = sum(x, 2);                % cost of population
ff_total_init = sum(ff_init);       % sum of fitness
average_ff_init = mean(ff_init)     % average fitness

ff_av_array = average_ff_init;
ff_max_array = max(ff_init);

ff = ff_init;
iterations = 0;
iter_array = iterations;

while iterations < 70

    [B,I] = sort(ff); % B - fitness sorted, I - corresponding indices

    selectedIndividuals = select_individuals(B);

    new_x = mate(selectedIndividuals, x, I, population);

    x = new_x;

    ff = sum(x, 2);
    ff_total = sum(ff);

    iterations = iterations + 1;

    ff_av_array = [ff_av_array mean(ff)];
    ff_max_array = [ff_max_array max(ff)];
    iter_array = [iter_array iterations];
end

ff;
average_ff = mean(ff)

figure(1)
subplot(2, 1, 1)
stem(iter_array, ff_av_array, 'r', 'filled', 'LineStyle', 'none')
xlabel('Iterations')
ylabel('Average fitness of population')
subplot(2, 1, 2)
stem(iter_array, ff_max_array, 'r', 'filled', 'LineStyle', 'none')
xlabel('Iterations')
ylabel('Maximum fitness in population')
```

Appendix 2 (`select-individuals.m`)

```
function [selectedIndividuals] = select_individuals(sorted_fit)

    probability = sorted_fit/sum(sorted_fit);

    for j = 1:length(probability)
        matingVector(j,1) = sum(probability(1:j));
    end

    for j = 1:length(probability)
        matingSelector = rand;

        for i = 1:length(probability)
            if matingSelector < matingVector(i,1)
                selectedIndividuals(j) = i;
                break
            end
        end
    end

end
```

Appendix 3

```
function new_population = mate(selectedIndividuals, x, I, population)

    i = population;
    nelem = size(x, 2);

    while i - 1 > 0

        crossPoint = round(rand*nelem);

        Indi1 = [x(I(selectedIndividuals(i)), 1:crossPoint)
x(I(selectedIndividuals(i-1)), crossPoint+1:nelem)];
        Indi2 = [x(I(selectedIndividuals(i-1)), 1:crossPoint)
x(I(selectedIndividuals(i)), crossPoint+1:nelem)];

        new_population(population + 1 - i, :) = Indi1;
        new_population(population + 2 - i, :) = Indi2;

        i = i - 2; % population size must be even for algorithm to work
properly
    end

end
```